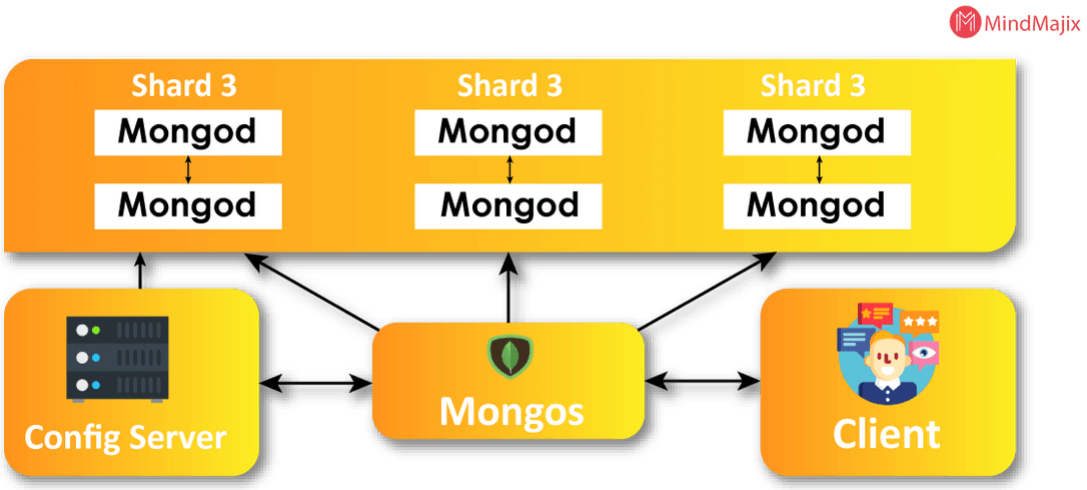
The architecture of MongoDB NoSQL Database

The following are the components of MongoDB architecture:

- Database
- Collection
- Document



Database

It is also called the physical container for data. Every database has its own set of files existing on the file system. In a single MongoDB server, there are multiple databases present.

Collection

The collection consists of various documents from different fields. All the collections reside within one database. In collections no schemas are present.

Document

The set of key values are assigned to the document which is in turn associated with dynamic schemas. The benefit of using these schemas is that a document may not have to possess the same fields whereas they can have different data types.

1. What is MongoDB, and How Does It Differ from Traditional SQL Databases?

MongoDB is a NoSQL database which means it does not use the **traditional table-based** relational database structure. Instead of it uses a flexible and document-oriented data model that stores data in **BSON** (Binary JSON) format.

Unlike SQL databases that use rows and columns, MongoDB stores data as JSON-like documents, making it easier to handle unstructured data and providing greater flexibility in terms of schema design.

2. Explain BSON and Its Significance in MongoDB.

BSON (Binary JSON) is a binary-encoded serialization format used by MongoDB to store documents. BSON extends JSON by adding support for data types such as dates and binary data and it is designed to be efficient in both storage space and scan speed. The binary format allows MongoDB to be more efficient with data retrieval and storage compared to text-based JSON.

3. Describe the Structure of a MongoDB Document.

A MongoDB document is a set of **key**-**value** pairs similar to a JSON object. Each key is a string and the value can be a variety of data types including strings, numbers, arrays, nested documents and more.

For example:

```
{
  "_id": ObjectId("507f1f77bcf86cd799439011"),
  "name": "Alice",
  "age": 25,
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "CA"
  },
  "hobbies": ["reading", "cycling"]
}
```

4. What are Collections And Databases In MongoDB?

In MongoDB, a database is a container for collections, and a collection is a group of MongoDB documents. Collections are equivalent to tables in SQL databases, but they do not enforce a schema allowing for flexibility in the structure of the documents they contain. For example, you might have a users collection in a mydatabase database.

5. How Does MongoDB Ensure High Availability and Scalability?

MongoDB ensures high **availability** and **scalability** through its features like **replica sets** and **sharding**. Replica sets provide redundancy and failover capabilities by ensuring that data is always available.

Sharding distributes data across multiple servers, enabling horizontal scalability to handle large volumes of data and high traffic loads.

6. Explain the Concept of Replica Sets in MongoDB.

A **replica set** in MongoDB is a group of mongod instances that maintain the same data set. A replica set consists of a primary node and multiple **secondary nodes**.

The primary node receives all write operations while secondary nodes replicate the primary's data and can serve read operations. If the primary node fails, an automatic election process selects a new primary to maintain high availability.

7. How replication works in MongoDB?

Write Operations: Write operations are sent to the primary node. The primary node records the changes in its oplog.

Replication: The secondary nodes replicate the primary's oplog and apply the changes to their data sets.

Failover: If the primary node fails, an election is held among the secondary nodes to choose a new primary. The new primary node then continues to handle write operations.

Recovery: If the failed primary node comes back online, it becomes a secondary node and starts replicating the new primary's Oplog

7. Explain about MongoDB architecture flow?

Client Interaction: Clients (applications or users) send requests to MongoDB for operations such as inserting, querying, updating, or deleting data.

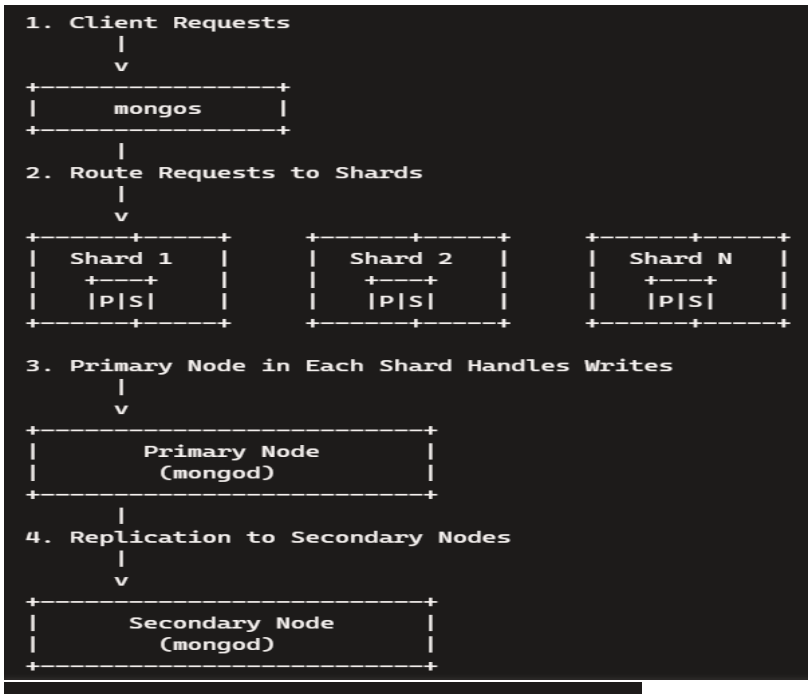
Query Routing: For sharded clusters, the mongos router receives client requests and routes them to the appropriate shards based on the shard key.

Data Storage: Insert Operations: The primary node in a replica set writes the data to its storage and records the operation in the oplog.

Query Operations: Queries are processed by the primary or secondary nodes, depending on the read preference settings. The node retrieves the requested data and returns it to the client.

Replication: Secondary nodes continuously replicate the primary's oplog and apply the operations to their data sets, ensuring data consistency across the replica set.

Sharding: Data is distributed across multiple shards based on the shard key. Each shard stores a portion of the overall data, enabling horizontal scaling and efficient data distribution.



Client Requests: Applications or users send data requests.

Mongos Router: Directs queries to the appropriate shards.

Shards: Store subsets of the data, with primary and secondary nodes ensuring data consistency and fault tolerance.

Config Servers: Manage metadata about the sharded cluster.

7. In MongoDB who will increase the shading?

In MongoDB, the task of increasing or managing sharding falls on the **cluster administrator** or **database administrator** (DBA). The cluster administrator is responsible for configuring, managing, and maintaining the sharded cluster.

Steps to Increase Sharding in MongoDB

Monitor Cluster Performance: Regularly monitor the performance of the sharded cluster to identify when additional shards are needed. Tools like MongoDB Atlas, Prometheus, and Grafana can help with this.

Add Shards to the Cluster: When it's determined that additional shards are needed, the cluster administrator will add new shards to the cluster. This involves provisioning new MongoDB instances (shards) and configuring them to join the existing cluster.

Add Shard Command: The cluster administrator uses the addShard command in the mongos router to add a new shard to the cluster

sh.addShard("hostname:port")

7. MongoDB default replication factor?

The default replication factor in MongoDB is 3. This means that, by default, each replica set in MongoDB consists of three members: one primary node and two secondary nodes. This setup ensures that there are multiple copies of the data for redundancy and high availability.

7. What are the Advantages of Using MongoDB Over Other Databases?

- **Flexibility**: MongoDB's document-oriented model allows for dynamic schemas.
- **Scalability**: Built-in sharding enables horizontal scaling.
- **High Availability**: Replica sets provide redundancy and automatic failover.
- **Performance**: Efficient storage and retrieval with BSON format.
- **Ease of Use**: JSON-like documents make it easier for developers to interact with data.

8. How to Create a New Database and Collection in MongoDB?

To create a new database and collection in MongoDB, you can use the mongo shell:

```
use mydatabase
db.createCollection("mycollection")
```

This command switches to mydatabase (creating it if it doesn't exist) and creates a new collection named mycollection.

9. What is Sharding, and How Does It Work in MongoDB?

Sharding is a method for distributing data across multiple servers in MongoDB. It allows for horizontal scaling by splitting large datasets into smaller, more manageable pieces called **shards**.

Each shard is a separate database that holds a portion of the data. MongoDB automatically balances data and load across shards, ensuring efficient data distribution and high performance.

10. Explain the Basic Syntax of MongoDB CRUD Operations.

CRUD operations in MongoDB are used to create, read, update, and delete documents.

- **Create: db.collection.insertOne({ name: "Alice", age: 25 })**
- **Read: db.collection.find({ name: "Alice" })**
- **Update: db.collection.updateOne({ name: "Alice" }, { $set: { age: 26 } })**
- **Delete: db.collection.deleteOne({ name: "Alice" })**

11. How to Perform Basic Querying in MongoDB?

Basic querying in MongoDB involves using the find method to retrieve documents that match certain criteria. For example:

db.collection.find({ age: { $gte: 20 } })

This query retrieves all documents from the collection where the age field is greater than or equal to 20.

12. What is an Index in MongoDB, and How to Create One?

An **index** in MongoDB is a data structure that improves the speed of data retrieval operations on a collection. You can create an index using the createIndex method.

For example, to create an index on the name field:

db.collection.createIndex({ name: 1 })

13. How Does MongoDB Handle Data Consistency?

MongoDB provides several mechanisms to ensure data consistency:

- **Journaling:** MongoDB uses write-ahead logging to maintain data integrity.
- **Write Concerns:** It specify the level of acknowledgment requested from MongoDB for write operations (e.g., acknowledgment from primary only, or acknowledgment from primary and secondaries).
- **Replica Sets:** Replication ensures data is consistent across multiple nodes, and read concerns can be configured to ensure data consistency for read operations.

14. How to Perform Data Import and Export in MongoDB?

To perform data import and export in MongoDB, you can use the mongoimport and mongoexport tools. These tools allow you to import data from **JSON, CSV or TSV** files into MongoDB and export data from MongoDB collections to JSON or CSV files.

Import Data:

mongoimport --db mydatabase --collection mycollection --file data.json

This command imports data from data.json into the mycollection collection in the mydatabase database.

Export Data:

mongoexport --db mydatabase --collection mycollection --out data.json

This command exports data from the mycollection collection in the mydatabase database to data.json.

15. What are MongoDB Aggregation Pipelines and How are They Used?

The **aggregation pipeline** is a framework for data aggregation, modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into aggregated results.

Each stage performs an operation on the input documents and passes the results to the next stage.

```
db.orders.aggregate([
  { $match: { status: "A" } },          // Stage 1: Filter documents by status
  { $group: { _id: "$cust_id", total: { $sum: "$amount" } } },  // Stage 2: Group by customer ID and sum the amount
  { $sort: { total: -1 } }              // Stage 3: Sort by total in descending order
])
```

In this example:

- Stage 1 ($match) filters documents by status "A".
- Stage 2 ($group) groups documents by customer ID and calculates the total amount for each group.
- Stage 3 ($sort) sorts the results by total amount in descending order.

Aggregation pipelines are powerful and flexible, enabling complex data processing tasks to be executed within MongoDB.

16. Describe the Aggregation Framework in MongoDB

The Aggregation Framework in MongoDB is a powerful tool for performing data processing and transformation on documents within a collection. It works by passing documents through a multi-stage pipeline, where each stage performs a specific operation on the data, such as **filtering, grouping, sorting, reshaping and computing aggregations.**

This framework is particularly useful for creating complex data transformations and analytics directly within the **database**.

17. How to Perform Aggregation Operations Using MongoDB?

Aggregation operations in MongoDB are performed using the aggregate method. This method takes an array of pipeline stages, each stage representing a step in the data processing pipeline. **For**

example, to calculate the total sales for each product, you might use the following aggregation pipeline:

```
db.sales.aggregate([
  { $match: { status: "completed" } },  // Filter completed sales
  { $group: { _id: "$product", totalSales: { $sum: "$amount" } } },  // Group by product and sum the sales amount
  { $sort: { totalSales: -1 } }  // Sort by total sales in descending order
])
```

18. Explain the Concept of Write Concern and Its Importance in MongoDB

Write Concern in MongoDB refers to the level of acknowledgment requested from MongoDB for write operations. It determines how many nodes must confirm the write operation before it is considered successful. Write concern levels range from "**acknowledged**" (default) to "**unacknowledged**," "**journaled**," and various "**replica acknowledged**" levels.

The importance of write concern lies in balancing between data durability and performance. Higher write concern ensures data is safely written to disk and replicated, but it may impact performance due to the added latency.

19. What are TTL Indexes, and How are They Used in MongoDB?

TTL (Time To Live) **Indexes** in MongoDB are special indexes that automatically remove documents from a collection after a certain period. They are commonly used for data that needs to expire after a specific time, such as **session information**, **logs**, or **temporary data**.

To create a TTL index, you can specify the expiration time in seconds:

```
db.sessions.createIndex({ "createdAt": 1 }, { expireAfterSeconds: 3600 })
```

This index will remove documents from the sessions collection 1 hour (3600 seconds) after the createdAt field's value.

20. How to Handle Schema Design and Data Modeling in MongoDB?

Schema design and **data modeling** in MongoDB involve defining how data is organized and stored in a document-oriented database. Unlike SQL databases, MongoDB offers flexible schema design, which can be both an advantage and a challenge. Key considerations for schema design include:

- **Embedding** vs. **Referencing**: Deciding whether to embed related data within a single document or use references between documents.
- **Document Structure**: Designing documents that align with application query patterns for efficient read and write operations.
- **Indexing**: Creating indexes to support query performance.
- **Data Duplication**: Accepting some level of data duplication to optimize for read performance.
- **Sharding**: Designing the schema to support sharding if horizontal scaling is required.

21. What is GridFS, and When is it Used in MongoDB?

GridFS is a specification for storing and retrieving large files in MongoDB. It is used when files exceed the BSON-document size limit of 16 MB or when you need to perform efficient retrieval of specific file sections.

GridFS splits a large file into smaller chunks and stores each chunk as a separate document within two collections: **fs.files** and **fs.chunks**. This allows for efficient storage and retrieval of large files, such as images, videos, or large datasets.

22. Explain the Differences Between WiredTiger and MMAPv1 Storage Engines

WiredTiger and **MMAPv1** are two different storage engines in MongoDB, each with distinct characteristics:

1. WiredTiger:

- **Concurrency**: Provides document-level concurrency, allowing multiple operations to be performed simultaneously.
- **Compression**: Supports data compression, reducing storage space requirements.
- **Performance**: Generally offers better performance and efficiency for most workloads.
- **Journaling**: Uses a write-ahead logging mechanism for better data integrity.

2. MMAPv1:

- **Concurrency**: Provides collection-level concurrency, which can limit performance under heavy write operations.
- No **Compression**: Does not support data compression.
- **Legacy Engine**: It is the older storage engine and has been deprecated in favor of WiredTiger.
- **Simplicity**: Simple implementation but lacks advanced features compared to WiredTiger.

23. How to Handle Transactions in MongoDB?

MongoDB supports multi-document ACID transactions by allowing us to perform a series of read and write operations across multiple documents and collections in a transaction.

This ensures data consistency and integrity. To use transactions we typically start a session, begin a transaction, perform the operations and then **commit** or **abort** the transaction.

Example in JavaScript:

```
const session = client.startSession();

session.startTransaction();

try {
  db.collection1.insertOne({ name: "Alice" }, { session });
  db.collection2.insertOne({ name: "Bob" }, { session });
  session.commitTransaction();
} catch (error) {
  session.abortTransaction();
} finally {
```

```
  session.endSession();
}
```

24. Describe the MongoDB Compass Tool and Its Functionalities

MongoDB Compass is a graphical user interface (GUI) tool for MongoDB that provides an easy way to visualize, explore, and manipulate your data. It offers features such as:

- **Schema Visualization**: View and analyze your data schema, including field types and distributions.
- **Query Building**: Build and execute queries using a visual interface.
- **Aggregation Pipeline**: Construct and run aggregation pipelines.
- **Index Management**: Create and manage indexes to optimize query performance.
- **Performance Monitoring**: Monitor database performance, including slow queries and resource utilization.
- **Data Validation**: Define and enforce schema validation rules to ensure data integrity.
- **Data Import/Export**: Easily import and export data between MongoDB and JSON/CSV files.

25. What is MongoDB Atlas, and How Does it Differ From Self-Hosted MongoDB?

MongoDB Atlas is a fully managed cloud database service provided by MongoDB. It offers automated deployment, scaling, and management of MongoDB clusters across various cloud providers (AWS, Azure, Google Cloud). Key differences from self-hosted MongoDB include:

- **Managed Service**: Atlas handles infrastructure management, backups, monitoring, and upgrades.
- **Scalability**: Easily scale clusters up or down based on demand.
- **Security**: Built-in security features such as encryption, access controls, and compliance certifications.
- **Global Distribution**: Deploy clusters across multiple regions for low-latency access and high availability.
- **Integrations**: Seamless integration with other cloud services and MongoDB tools.

26. How to Implement Access Control and User Authentication in MongoDB?

Access control and user authentication in MongoDB are implemented through a role-based access control (RBAC) system. You create users and assign roles that define their permissions. To set up access control:

- **Enable Authentication**: Configure MongoDB to require authentication by starting the server with --auth or setting security.authorization to enabled in the configuration file.
- Create **Users**: Use the db.createUser method to create users with specific roles.

```
db.createUser({
  user: "admin",
  pwd: "password",
```

```
  roles: [{ role: "userAdminAnyDatabase", db: "admin" }]
});
```

- **Assign Roles:** Assign roles to users that define their permissions, such as read, write, or admin roles for specific databases or collections.

27. What are Capped Collections, and When are They Useful?

Capped collections in MongoDB are fixed-size collections that automatically overwrite the oldest documents when the specified size limit is reached. They maintain insertion order and are useful for scenarios where you need to store a fixed amount of recent data, such as logging, caching, or monitoring data.

Example of creating a capped collection:

```
db.createCollection("logs", { capped: true, size: 100000 });
```

28. Explain the Concept of Geospatial Indexes in MongoDB

Geospatial indexes in MongoDB are special indexes that support querying of geospatial data, such as locations and coordinates. They enable efficient queries for proximity, intersections, and other spatial relationships. MongoDB supports two types of geospatial indexes: **2d** for **flat** geometries and **2dsphere** for spherical geometries.

Example of creating a 2dsphere index:

```
db.places.createIndex({ location: "2dsphere" });
```

29. How to Handle Backups and Disaster Recovery in MongoDB?

Handling backups and disaster recovery in MongoDB involves regularly creating backups of your data and having a plan for restoring data in case of failure. Methods include:

- **Mongodump/Mongorestore**: Use the mongodump and mongorestore utilities to create and restore binary backups.
- **File System Snapshots**: Use file system snapshots to take consistent backups of the data files.
- **Cloud Backups**: If using MongoDB Atlas, leverage automated backups provided by the service.
- **Replica Sets**: Use replica sets to ensure data redundancy and high availability. Regularly test the failover and recovery process.

30. Describe the Process of Upgrading MongoDB to a Newer Version

Upgrading MongoDB to a newer version involves several steps to ensure a smooth transition:

- **Check Compatibility**: Review the release notes and compatibility changes for the new version.
- **Backup Data**: Create a backup of your data to prevent data loss.
- **Upgrade Drivers**: Ensure that your application drivers are compatible with the new MongoDB version.

- Upgrade MongoDB: Follow the official MongoDB upgrade instructions, which typically involve stopping the server, installing the new version, and restarting the server.
- **Test Application**: Thoroughly test your application with the new MongoDB version to identify any issues.
- **Monitor**: Monitor the database performance and logs to ensure a successful upgrade.

31. What are Change Streams in MongoDB, and How are They Used?

Change Streams in MongoDB allow applications to listen for real-time changes to data in collections, databases, or entire clusters. They provide a powerful way to implement event-driven architectures by capturing insert, update, replace, and delete operations.

To use Change Streams, you typically open a change stream cursor and process the change events as they occur.

Example:

```
const changeStream = db.collection('orders').watch();
changeStream.on('change', (change) => {
  console.log(change);
});
```

This example listens for changes in the orders collection and logs the change events.

32. Explain the Use of Hashed Sharding Keys in MongoDB

Hashed Sharding Keys in MongoDB distribute data across shards using a hashed value of the shard key field. This approach ensures an even distribution of data and avoids issues related to data locality or uneven data distribution that can occur with range-based sharding.

Hashed sharding is useful for fields with monotonically increasing values, such as timestamps or identifiers.

Example:

```
db.collection.createIndex({ _id: "hashed" });
sh.shardCollection("mydb.mycollection", { _id: "hashed" });
```

33. How to Optimize MongoDB Queries for Performance?

Optimizing MongoDB queries involves several strategies:

- **Indexes:** Create appropriate indexes to support query patterns.
- **Query Projections:** Use projections to return only necessary fields.
- **Index Hinting:** Use index hints to force the query optimizer to use a specific index.
- **Query Analysis:** Use the explain() method to analyze query execution plans and identify bottlenecks.
- **Aggregation Pipeline:** Optimize the aggregation pipeline stages to minimize data processing and improve efficiency.

34. Describe the Map-Reduce Functionality in MongoDB

Map-Reduce in **MongoDB** is a data processing paradigm used to perform complex data aggregation operations. It consists of two phases: the map phase processes each input document and emits key-value pairs, and the reduce phase processes all emitted values for each key and outputs the final result.

Example:

```
db.collection.mapReduce(
  function() { emit(this.category, this.price); },
  function(key, values) { return Array.sum(values); },
  { out: "category_totals" }
);
```

This example calculates the total price for each category in a collection.

35. What is the Role of Journaling in MongoDB, and How Does It Impact Performance?

Journaling in MongoDB ensures data durability and crash recovery by recording changes to the data in a journal file before applying them to the database files. This mechanism allows MongoDB to recover from unexpected shutdowns or crashes by replaying the journal.

While journaling provides data safety, it can impact performance due to the additional I/O operations required to write to the journal file.

36. How to Implement Full-Text Search in MongoDB?

Full-Text Search in MongoDB is implemented using text indexes. These indexes allow you to perform text search queries on string content within documents.

Example:

```
db.collection.createIndex({ content: "text" });
db.collection.find({ $text: { $search: "mongodb" } });
```

In this example, a text index is created on the content field, and a text search query is performed to find documents containing the word "mongodb."

37. What are the Considerations for Deploying MongoDB in a Production Environment?

Considerations for deploying MongoDB in a production environment include:

- **Replication**: Set up replica sets for high availability and data redundancy.
- **Sharding**: Implement sharding for horizontal scaling and to distribute the load.
- **Backup and Recovery**: Establish a robust backup and recovery strategy.
- **Security**: Implement authentication, authorization, and encryption.
- **Monitoring**: Use monitoring tools to track performance and detect issues.
- **Capacity Planning**: Plan for adequate storage, memory, and CPU resources.
- **Maintenance**: Regularly update MongoDB to the latest stable version and perform routine maintenance tasks.

38. Explain the Concept of Horizontal Scalability and Its Implementation in MongoDB

Horizontal Scalability in MongoDB refers to the ability to add more servers to distribute the load and data. This is achieved through sharding, where data is partitioned across multiple shards.

Each shard is a replica set that holds a subset of the data. Sharding allows MongoDB to handle large datasets and high-throughput operations by distributing the workload.

39. How to Monitor and Troubleshoot Performance Issues in MongoDB?

Monitoring and troubleshooting performance issues in MongoDB involve:

- **Monitoring Tools**: Use tools like MongoDB Cloud Manager, MongoDB Ops Manager, or third-party monitoring solutions.
- **Logs**: Analyze MongoDB logs for errors and performance metrics.
- **Profiling**: Enable database profiling to capture detailed information about operations.
- **Explain Plans**: Use the explain() method to understand query execution and identify bottlenecks.
- **Index Analysis**: Review and optimize indexes based on query patterns and usage.
- **Resource Utilization**: Monitor CPU, memory, and disk I/O usage to identify resource constraints.

40. Describe the Process of Migrating Data from a Relational Database to MongoDB

Migrating data from a relational database to MongoDB involves several steps:

- **Schema Design**: Redesign the relational schema to fit MongoDB's document-oriented model. Decide on embedding vs. referencing, and plan for indexes and collections.
- **Data Export**: Export data from the relational database in a format suitable for MongoDB (e.g., CSV, JSON).
- **Data Transformation**: Transform the data to match the MongoDB schema. This can involve converting data types, restructuring documents, and handling relationships.
- **Data Import**: Import the transformed data into MongoDB using tools like mongoimport or custom scripts.
- **Validation**: Validate the imported data to ensure consistency and completeness.
- **Application Changes**: Update the application code to interact with MongoDB instead of the relational database.
- **Testing**: Thoroughly test the application and the database to ensure everything works as expected.
- **Go Live**: Deploy the MongoDB database in production and monitor the transition.

MongoDB Query Based Interview Questions

MongoDB Query-Based Interview Questions focus on your ability to write and optimize queries to interact with the database effectively.

These questions typically include tasks like retrieving specific data using filters, sorting and paginating results, and using projection to select fields.

```
"[
  {
    ""_id"": 1,
    ""name"": ""John Doe"",
    ""age"": 28,
    ""position"": ""Software Engineer"",
    ""salary"": 80000,
    ""department"": ""Engineering"",
    ""hire_date"": ISODate(""2021-01-15"")
  },
  {
    ""_id"": 2,
    ""name"": ""Jane Smith"",
    ""age"": 34,
    ""position"": ""Project Manager"",
    ""salary"": 95000,
    ""department"": ""Engineering"",
    ""hire_date"": ISODate(""2019-06-23"")
  },
  {
    ""_id"": 3,
    ""name"": ""Emily Johnson"",
    ""age"": 41,
    ""position"": ""CTO"",
    ""salary"": 150000,
    ""department"": ""Management"",
    ""hire_date"": ISODate(""2015-03-12"")
  },
  {
    ""_id"": 4,
    ""name"": ""Michael Brown"",
    ""age"": 29,
    ""position"": ""Software Engineer"",
    ""salary"": 85000,
    ""department"": ""Engineering"",
    ""hire_date"": ISODate(""2020-07-30"")
  },
  {
    ""_id"": 5,
    ""name"": ""Sarah Davis"",
    ""age"": 26,
    ""position"": ""UI/UX Designer"",
    ""salary"": 70000,
    ""department"": ""Design"",
    ""hire_date"": ISODate(""2022-10-12"")
  }
]"
```

1. Find all Employees Who Work in the "Engineering" Department.

Query:

```
db.employees.find({ department: "Engineering" })
```

Output:

```
[
  {
    "_id": 1,
    "name": "John Doe",
    "age": 28,
    "position": "Software Engineer",
    "salary": 80000,
    "department": "Engineering",
    "hire_date": ISODate("2021-01-15")
  },
  {
    "_id": 2,
    "name": "Jane Smith",
    "age": 34,
    "position": "Project Manager",
    "salary": 95000,
    "department": "Engineering",
    "hire_date": ISODate("2019-06-23")
  },
  {
    "_id": 4,
    "name": "Michael Brown",
    "age": 29,
    "position": "Software Engineer",
    "salary": 85000,
    "department": "Engineering",
    "hire_date": ISODate("2020-07-30")
  }
]
```

Explanation:

This query finds all employees whose department field is "Engineering".

2. Find the Employee with the Highest Salary.

Query:

```
db.employees.find().sort({ salary: -1 }).limit(1)
```

Output:

```
[
  {
    "_id": 3,
```

```
    "name": "Emily Johnson",
    "age": 41,
    "position": "CTO",
    "salary": 150000,
    "department": "Management",
    "hire_date": ISODate("2015-03-12")
  }
]
```

Explanation:

This query sorts all employees by salary in descending order and retrieves the top document, which is the employee with the highest salary.

3. Update the Salary of "John Doe" to 90000.

Query:

```
db.employees.updateOne({ name: "John Doe" }, { $set: { salary: 90000 } })
```

Output:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

Explanation:

This query updates the salary of the employee named "John Doe" to 90000.

4. Count the Number of Employees in Each Department.

Query:

```
db.employees.aggregate([
  { $group: { _id: "$department", count: { $sum: 1 } } }
])
```

Output:

```
[
  { "_id": "Engineering", "count": 3 },
  { "_id": "Management", "count": 1 },
  { "_id": "Design", "count": 1 }
]
```

Explanation:

This query groups the employees by the department field and counts the number of employees in each department.

5. Add a New Field Bonus to All Employees in the "Engineering" Department with a Value of 5000.

Query:

```
db.employees.updateMany({ department: "Engineering" }, { $set: { bonus: 5000 } })
```

Output:

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }

Explanation:

This query adds a new field bonus with a value of 5000 to all employees in the "Engineering" department.

6. Retrieve All Documents in the Employees Collection and Sort Them by the Length of Their Name in Descending Order.

Query:

```
db.employees.aggregate([
  { $addFields: { nameLength: { $strLenCP: "$name" } } },
  { $sort: { nameLength: -1 } },
  { $project: { nameLength: 0 } }
])
```

Output:

```
[
  {
    "_id": 2,
    "name": "Jane Smith",
    "age": 34,
    "position": "Project Manager",
    "salary": 95000,
    "department": "Engineering",
    "hire_date": ISODate("2019-06-23")
  },
  {
    "_id": 3,
    "name": "Emily Johnson",
    "age": 41,
    "position": "CTO",
    "salary": 150000,
    "department": "Management",
    "hire_date": ISODate("2015-03-12")
  },
  {
    "_id": 1,
    "name": "John Doe",
    "age": 28,
    "position": "Software Engineer",
    "salary": 80000,
    "department": "Engineering",
    "hire_date": ISODate("2021-01-15")
  },
  {
    "_id": 4,
    "name": "Michael Brown",
```

```
    "age": 29,
    "position": "Software Engineer",
    "salary": 85000,
    "department": "Engineering",
    "hire_date": ISODate("2020-07-30")
  },
  {
    "_id": 5,
    "name": "Sarah Davis",
    "age": 26,
    "position": "UI/UX Designer",
    "salary": 70000,
    "department": "Design",
    "hire_date": ISODate("2022-10-12")
  }
]
```

Explanation:

This query calculates the length of each employee's name, sorts the documents by this length in descending order, and removes the temporary nameLength field from the output.

7. Find the Average Salary of Employees in the "Engineering" Department.

Query:

```
db.employees.aggregate([
  { $match: { department: "Engineering" } },
  { $group: { _id: null, averageSalary: { $avg: "$salary" } } }
])
```

Output:

```
[
  { "_id": null, "averageSalary": 86666.66666666667 }
]
```

Explanation:

This query filters employees to those in the "Engineering" department and calculates the average salary of these employees.

8. Find the Department with the Highest Average Salary.

Query:

```
db.employees.aggregate([
  { $group: { _id: "$department", averageSalary: { $avg: "$salary" } } },
  { $sort: { averageSalary: -1 } },
  { $limit: 1 }
])
```

Output:

```
[
  { "_id": "Management", "averageSalary": 150000 }
]
```

Explanation:

This query groups employees by department, calculates the average salary for each department, sorts these averages in descending order, and retrieves the department with the highest average salary.

9. Find the Total Number of Employees Hired in Each Year.

Query:

```
db.employees.aggregate([
  { $group: { _id: { $year: "$hire_date" }, totalHired: { $sum: 1 } } }
])
```

Output:

```
[
  { "_id": 2015, "totalHired": 1 },
  { "_id": 2019, "totalHired": 1 },
  { "_id": 2020, "totalHired": 1 },
  { "_id": 2021, "totalHired": 1 },
  { "_id": 2022, "totalHired": 1 }
]
```

Explanation:

This query groups employees by the year they were hired, which is extracted from the hire_date field, and counts the total number of employees hired each year.

10. Find the Highest and Lowest Salary in the "Engineering" Department.

Query:

```
db.employees.aggregate([
  { $match: { department: "Engineering" } },
  {
    $group: {
      _id: null,
      highestSalary: { $max: "$salary" },
      lowestSalary: { $min: "$salary" }
    }
  }
])
```

Output:

```
[
  { "_id": null, "highestSalary": 95000, "lowestSalary": 80000 }
]
```

Explanation:

This query filters employees to those in the "Engineering" department, then calculates the highest and lowest salary within this group.

Q) Spring boot Mongo db example?

Spring Boot + MongoDB CRUD Example Tutorial

Q) How to install Mongo Atlas?

Spring Boot - Build a CRUD REST API with MongoDB Atlas | JavaTechie